Treebeard : An Optimizing Compiler for Tree Based ML Inference

ABSTRACT

Decision tree ensembles are commonly used machine learning models generated by machine learning techniques like gradient boosting and random forests. These models are used in a wide range of applications and are deployed at scale (run repeatedly on billions of data items). Several libraries such as XGBoost, LightGBM, and Sklearn expose algorithms for both training and inference with decision tree ensembles. While these libraries incorporate a limited set of hardware-specific optimizations, they do not specialize the inference code to the model being used, leaving significant performance on the table.

This paper presents a compiler-based approach that automatically generates efficient code for decision tree inference. It develops Treebeard, an extensible compiler, which progressively lowers inference code to LLVM IR through multiple intermediate abstractions. By applying model-specific optimizations at the higher levels, loop optimizations at the middle level, and machine-specific optimizations lower down, Treebeard can specialize inference code for each model on each supported hardware target. To improve model inference performance, Treebeard performs several novel optimizations such as tree tiling, tree walk unrolling, and tree walk interleaving.

We implement Treebeard using the MLIR compiler infrastructure and demonstrate the utility of Treebeard by evaluating it on a diverse set of tree ensemble models. Experimental evaluation demonstrates that Treebeard optimizations improve average latency over a batch of inputs by 2.2X over our baseline version. Further, Treebeard is significantly faster than XGBoost and Treelite in both single-core (2.8X and 5.1X respectively) and multi-core (3.2X and 2.6X respectively) settings.

1. INTRODUCTION

Intro tex here!

2. COMPILER OVERVIEW

Figure 1 shows the high level structure of Treebeard. The input to Treebeard is a serialized decision tree ensemble. Popular frameworks like XGBoost and LightGBM are supported and the system is easily extensible to other frameworks. Given an input model our compiler generates optimized inference code. Specifically it generates a callable batch inference function `predictForest` that given an array of input rows, outputs an array of model predictions.

Treebeard specializes the code generated for inference by progressively optimizing and lowering a high level representation of the `predictForest` function down to LLVM IR [1]. By ***lowering***, we mean the process of transforming an operation at a higher abstraction to a sequence of operations at a lower abstraction. Optimizations in Treebeard are implemented using a combination of annotation and lowering. An Operation at a higher abstraction is annotated with meta-data that indicates what kind of optimization is to be performed while lowering it. The lowering transformation uses this information to generate optimized lower level code. For example, tree tiling and loop order are decided at the highest abstraction. These decisions are communicated to the lowering pass which explicitly encodes them in the lowered IR as shown in figure **??**.

Treebeard's IR has three abstractions as shown in Figure **??**. At the highest level (HIR), the input model is represented as a collection of binary trees. This is shown in the second row of figure **??**. At this level of abstraction, Treebeard tiles nodes together to transform a binary tree into an n-ary tree and decides what order trees are to be traversed in. In the example in the figure, trees are tiled with a tile size of 2. Tiling is indicated using colored elipses drawn around nodes that are in the same tile. Tree reordering is another transformation that is performed at this abstraction. One objective of tree reordering is to group identically structured trees so that they can share the same traversal code. With the tiling shown in Figure **??**, Tree1 and Tree3 have depth 2, while Tree2 has a depth of 3. Therefore, the trees are reordered so that Tree1 and Tree3 are together.

The high level IR is then lowered to a mid-level IR (MIR). The aim of MIR is to allow optimizations that are independent of the final memory layout of the model. The lowering to MIR performs loop optimizations like loop-reordering, parallelization and unrolling. At this level of the IR (the third row in Figure **??**), the order in which trees, input row pairs are walked is explicitly represented in the IR using loop nests. Figure **??** shows two possible ways that loop nests could be generated – the first walks all rows for one tree before moving to the next tree while the second walks all trees for one row before moving to the next row. Another aspect handled by this lowering is the fissing of loops to make sure that trees with the same structure share code. Here, both versions of MIR shown ensure that Tree1 and Tree3 share the same traversal code, while traversal code for Tree2 is different. Additionally, tree walk optimizations, such as tree walk unrolling (shown

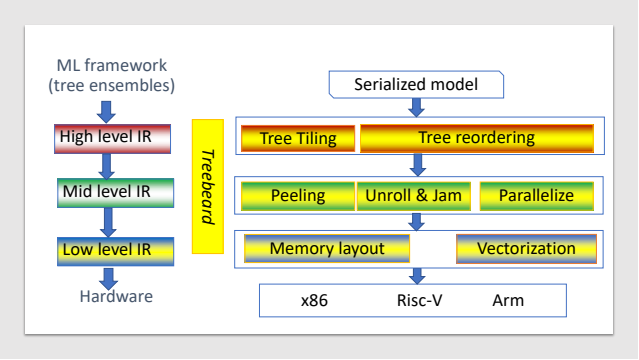

Figure 1: Treebeard Compiler Structure

in figure **??**) and peeling are performed on the MIR.

Treebeard then further lowers the IR to explicitly represent the memory layout of the model. Buffers to hold model values are inserted into the generated code and all tree operations in the mid-level IR are lowered to explicitly reference these buffers. Additionally, at this level of the IR, Treebeard generates vectorized code to take advantage of SIMD instructions where applicable. Finally, the inference function is translated to LLVM IR and JIT compiled to executable code.

3. OPTIMIZATIONS

TODO This section needs a better name!

3.1 Notation

TODO Notation needs to be introduced in the background section We represent a decision tree by $T = (V, E, r)$ where V is the set of nodes, E the set of edges and $r \in V$ is the root. For each node $n \in V$, the following are given.

1. $threshold(n) \in \mathbb{R}$ which gives the threshold value for n.
2. $featureIndex(n) \in \mathbb{N}$ which gives the feature index for n.
3. $left(n) \in V$, the left child of n or $\emptyset$ if n is a leaf. If $left(n) \neq \emptyset$, then $(n, left(n)) \in E$.
4. $right(n) \in V$, the right child of n or $\emptyset$ if n is a leaf. If $right(n) \neq \emptyset$, then $(n, right(n)) \in E$.

We use $L \subset V$ to denote the set of leaves.

3.2 Tiling

Treebeard vectorizes tree walks by grouping nodes of a decision tree into ***tiles***. The nodes in a tile are evaluated concurrently using vector instructions. Once the nodes of the current tile are evaluated, a look up table is used to compute which child of the current tile to move to next. The listing below shows at a high level how a tiled tree is walked.

```
1  // A lookup table that determines the child index of
2  // the next tile given the tile shape and the outcome
3  // of the vector comparison on the current tile
4  int16_t LUT[NUM_TILE_SHAPES, pow(2, TileSize)]
5
```

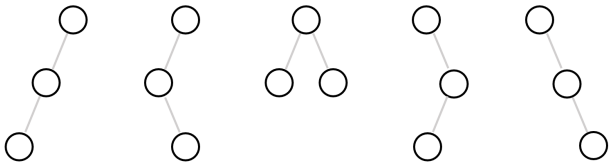

Figure 2: All possible tile shapes with tile size $n_t = 3$

```
6  ResultType Prediction_Function(...) {
7    // ...
8    Node n = getRoot(tree)
9    while (isLeaf(tree, n)==false) do {
10     thresholds = loadThresholds(tree, n)
11     featureIndices = loadFeatureIndices(tree, n)
12     // Gather required feature from the current row
13     features = rows[i][featureIndices]
14     // Vector comparison of features and thresholds
15     comparison = features < thresholds
16
17     // Pack bits in comparison vector into an integer
18     comparisonIndex = combineBitsIntoInt(comparison)
19
20     // Get child index of tile we need to move to
21     tileShape = loadTileShape(tree, n)
22     childIndex = LUT[tileShapeID, comparisonIndex]
23
24     // Move to the correct child of the current node
25     n = getChildNode(tree, n, childIndex)
26   }
27   ThresholdType prediction = getLeafValue(n)
28   // ...
29 }
```

To evaluate the current tile, the vector of thresholds is first loaded (`loadThresholds`). This vector contains the thresholds of all nodes in the tile. Then, the features required for comparison are gathered into a vector (lines 11 and 13). The feature vector is compared to the threshold vector and the child tile to move to is determined (lines 15 to 25). More details about tile shapes and the look up table are presented in subsequent sections.

3.3 Tiles and Tile Shapes

A ***tile*** is a collection of connected non-leaf nodes of a decision tree. The path connecting any pair of nodes in the tile must fully be contained within the tile. The tile size n_t is the number of nodes contained in a tile.

Informally, the ***tile shape*** is the shape of the region that encloses all nodes in a tile in a diagram of the decision tree. More formally, for a tile size n_t, each unique legal binary tree containing n_t nodes (nodes being indistinguishable) corresponds to a tile shape.

Figure 2 enumerates all tile shapes with a tile size of 3. There are a total of 5 tile shapes with size 3. The number of tile shapes with a tile size n_t, denoted by $NTS(n_t)$ is given by the following equation.

$$NTS(n) = \sum_{k=0}^{n-1} NTS(k) \times NTS(n-k-1) \qquad (1)$$

where $NTS(0) = NTS(1) = 1$.

3.4 Valid Tiling of a Tree

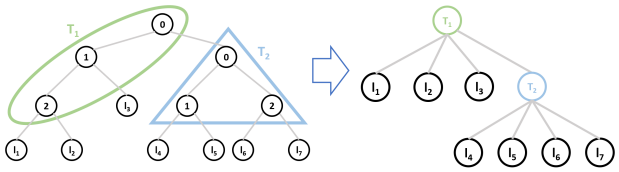

Figure 3: Example of a valid tree tiling with tile size $n_t = 3$

A tiling $\mathscr{T}$ of the tree $T = (V, E, r)$ with tile size n_t is a partition $\{T_1, T_2, ..., T_m\}$ of the set V such that

1. $T_1 \cup T_2 ... \cup T_m = V$
2. $T_i \cap T_j = \emptyset$ for all $i, j \in [1, m]$ and $i \neq j$
3. $|T_i| \leq n_t$ for all $i \in [1, m]$
4. $\forall l \in L : l \in T_i \rightarrow v \notin T_i \ \ \forall v \in V \backslash \{l\}$
5. Tiles are **maximal**, i.e. if $|T_i| < n_t$, then there is no $v \in V \backslash \{T_i \cup L\}$ such that $(u, v) \in E$ for some $u \in T_i$.
6. Tiles are **connected**, i.e. for an $u, v \in T_i$, there is a (undirected) path connecting u and v fully contained in T_i.

3.5 Tiled Trees

A tiling transformation communicates the tiling to the Treebeard infrastructure by assigning a tile ID to each node in the decision tree. Using these tile IDs, Treebeard checks the validity of the tiling and then contructs a tree whose nodes are tiles. We call this tree the ***tree of tiles***. TODO We need a better name for this Figure 3 shows a valid tiling with tile size 3 and the tree of tiles constructed by Treebeard. Three nodes are grouped into each of the tiles t_1 and t_2 as shown. Each tile is collapsed into a single node in the tree of tiles. However, each leaf in the original tree becomes a leaf in the tree of tiles.

Treebeard maintains the following invariants.

1. All tiles in a tree are the same size n_t. If the tiling produces any smaller tiles, these are padded by inserting dummy nodes to make them the required size.
2. Nodes within tiles are always ordered in level order and left to right within a level. The numbering of the nodes in the above diagram shows this node order.
3. Children of a node are numbered from left to right (regardless of level). For example, l_1 is the first child of t_1, l_2 is the second and so on.

3.6 Tile Shapes and Decision Tree Inference

Treebeard uses vector instructions to accelerate decision tree walks. Vector instructions are used to evaluate the predicates of all the nodes in a tile simultaneously. However, once the predicates of all the nodes in the tile are evaluated, computing the next tile to move to, given the outcome of the comparison depends on the tile shape of the current tile. To illustrate this problem, consider the case of the tiles of size 3 shown in figure 4. The diagram shows 3 of the 5 possible tile shapes for a tile size of 3. The nodes drawn in black are members of the tile t_1. The nodes in blue are the entry nodes of the children tiles of t_1. TODO Define entry nodes

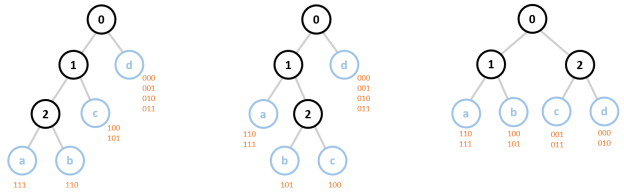

Figure 4: Example tile traversals with tile size $n_t = 3$

To traverse a tile on an input row, first, the predicate of each node in the tile is computed. Subsequently, we need to determine which of the child tiles to move to next. Note that a true predicte (bit value 1) on a node implies a move to the left child and a false predicate (bit value 0) implies a move to the right child.

In the diagram, the bit strings (written in red) show which child we need to move to given the outcomes of the comparison (the bits represent the comparison outcomes of nodes and are in the order of the nodes in the tile – marked 0, 1 and 2 in the diagram, i.e., the MSB is the predicate outcome of node 0 and the LSB the predicate outcome of node 2). For example, for the first tile shape, if the predicate of all nodes are true (i.e. the comparison outcome is 111), the next node to evaluate is a. However, if the predicate of node 1 is false, then we need to move to d regardless of the outcomes of nodes 2 and 3. It is easy to see from the diagram that, depending on the tile shape, the same predicate outcomes can mean moving to different children. For example, for the outcome "011", the next tile is the 4th child (node d) for the first two tile shapes while it is the 3rd child for the other tile shape (node c).

3.7 Lookup Table

A lookup table (LUT) is used to solve the problem described in section 3.6, i.e. given the outcome of the comparisons on all nodes in a tile, determine the child tile we should evaluate next. The LUT is indexed by the tile shape and the comparison outcome. Formally, the LUT is a map.

$$LUT : (TileShape, < Boolean \times n_t >) \rightarrow [0, n_t] \subset \mathbb{N}$$

where n_t is the tile size, $< Boolean \times n_t >$ is a vector of n_t booleans. The value returned by the LUT is the index of the child of the current tile that should be evaluated next. For example, if we are evaluating the first tile t in figure 4, and the result of the comparison is 110, then $LUT(TileShape(t), 110) = 1$ since the tile we need to evaluate next is the tile with node b, which is the second child of the current tile.

In order to realize this LUT in generated code, Treebeard associates a non-negative integer ID with every unique tile shape of the given tile size. The result of the comparison, a vector of booleans, can be interpreted as a 64-bit integer. Therefore, the LUT can be implemented as a 2 dimensional array.

```
int16_t LUT[NTS(n_t), pow(2, n_t)]
```

Treebeard computes the values in the LUT statically as the tile size is a compile time constant.

3.8 In Memory Representation of Tiled Trees

Treebeard currently has two in memory representations for tiled trees - an array based representation and a sparse representation. Both representations use an array of structs to represent all tiles of the model.

3.8.1 *Array Based Representation*

Each tree in the model is represented as an array of tiles using the standard representation of trees as arrays. The root node is at index 0 and for a node at index n in the array, the index of its i^{th} child is given by $(n_t+1)n+(i+1)$ (every node in the tree of tiles has n_t+1 children). A tile is represented by an object of the following struct.

```
struct Tile {
  // A vector of TileSize elements
  <ThresholdType x TileSize> thresholds;
  <FeatureIndexType x TileSize> featureIndices;
  // Integer that identifies the tile shape
  TileShapeIDType tileShapeID;
};
```

TODO AP Is this level of detail really needed? Also, the vector type notation needs to be introduced somewhere. Even though this representation is simple, the memory required for reasonable sized models is very large. The memory footprint is up to 20X that of the scalar representation. This memory bloat causes a performance degradation because the span of the L1 TLB is not sufficient to efficiently translate addresses for the whole model. TODO AP There are also some cache misses. Write this better. Storing leaves as full tiles (even though leaves just have to represent one value) and the empty space introduced due to the array based representation of trees that are not complete account for most of the increase. The sparse representation described next tries to address these issues.

3.8.2 *Sparse Representation*

The sparse representation tries to address the large memory footprint of the array based representation by doing the following.

- To eliminate the wasted space in the array representation, we add a child pointer to each tile. This points to the first child of the tile. All children of a tile are stored contiguously.
- Leaves are stored in a separate array. We found that, across all our benchmarks, a large fraction of leaves are such that all their siblings are also leaves. Such leaves are directly moved into the leaves array. For leaves for which this property does not hold, an additional hop is added by making the leaf tile a comparison tile and all its children are made leaves with the same value as the original leaf.

Figure 5 shows some of the details of the sparse representation. The tree on the left of the diagram is the actual decision

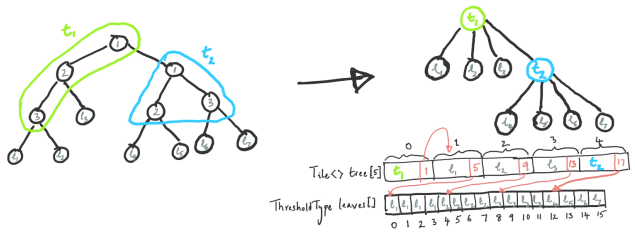

Figure 5: Sparse representation with tile size $n_t = 3$

tree with the nodes grouped into tiles t_1 and t_2. The tree on the right is the tree of tiles. The arrays depicted below show how the tree is represented in memory. The first array (`tree`) is an array of tiles and has 5 elements. Each element of the array represents a single tile and has the thresholds of the nodes, the feature indices, a tile shape ID and a pointer to the first child (shown explicitly in red).

As a specific example, consider the tile t_1. The tile has four children – l_1, l_2, l_3 and t_2 in that order (left to right). These tiles are stored contiguously in the `tree` array and a pointer to the first of these, l_1 is stored in the tile t_1 (the index 1 is stored in the tile t_1 as shown).

Now consider the tile t_2. Since all children of the tile t_2 are leaves, they are all moved into the `leaves` array. To store a pointer into the `leaves` array, we add `len(tree)` to the element index in the `leaves` array. The tile t_2's child is the element at index 12 of the `leaves` array. Therefore, the index $12+5=17$ is stored in the tile t_2. (Any index i that is greater than the length of the `tree` array is regarded as an index into the `leaves` array. The index into the `leaves` array is $i-$`len(tree)`.)

The other aspect of the representation is that an extra hop is added for the leaves l_1, l_2 and l_3 in order to simplify code generation. This enforces the invariant that all leaves are stored in the leaves array and simplifies checking whether we've reached a leaf. Therefore, 4 new leaves are added as children for each of the original leaves l_1, l_2 and l_3. Each of these 12 newly added leaves has the same value as its parent. These are the first 12 elements of the `leaves` array.

Even though we currently have implementations of the two representations detailed in sections 3.8.1 and 3.8.2, support for other representations is not hard to add. All optimizing passes that work on the high level and mid level IR will continue to work as is. Programmers need only provide new lowering passes for a few operations in the low level IR.

3.9 Uniform Tiling

The first tree tiling algorithm we implement is uniform tiling. The basic idea of uniform tiling is to construct tiles with a tile size n_t so that we evaluate the least possible number of nodes speculatively. Intuitively, this translates to minimizing the height of the constructed tile. We minimize the height of the tile by performing a level order traversal starting at the root node of the tile. TODO AP Should we say this is optimal when probabilities are equal? Problem is we haven't said what optimal is. This traversal collects at most n_t nodes and does not include any leaves in the tile so that the constraints listed in section 3.4 are respected. Algorithm 1 lists

the algorithm for uniform tiling. Tiling starts at the root and constructs a tile *Tile* by performing a level order traversal. Once the current tile is constructed, the tiling procedure is recursively performed on all nodes that are destinations for edges going out of the constructed tile. The set of tiles constructed by algorithm 1 constitutes a uniform tiling of the input tree.

Algorithm 1 Uniform tree tiling

procedure TILETREE($T = (V, E, r), n_t$)
  if $r \in L$ **then**
    return $\{r\}$
  end if
  //Level order traversal to collect n_t or fewer nodes.
  //Leaves are not included in the constructed tile.
  $Tile \leftarrow LevelOrderTraveral(r, n_t)$
  $Tiles = \{Tile\}$
  for $(u, v) \in Out(Tile)$ **do**
    $Tiles \leftarrow Tiles \cup TileTree(T_v, n_t)$
  end for
  return *Tiles*
end procedure

3.9.1 Further Opimization and Code Generation

We found that most leaf tiles for a given tree are at the same depth when uniform tiling is used. Furthermore, we see that deeper leaves are more likely to be reached. Based on these observations, we pad the tree of tiles generated with uniform tiling so that all leaves are at the same depth. This transformation is performed on the high level IR after uniform tiling. Once the trees have been padded to make all leaves equal depth, the tree walks are fully unrolled to evaluate a fixed number of tiles and all leaf checks are elided.

One other complication the code generator needs to handle is the fact that trees potentially have different depths. In order to handle this, Treebeard sorts the trees by their depth. This ensures that all trees with equal depth are grouped together. Once this is done, the loop over the trees is fissed so that each of the resulting loops only walks trees of a single depth. Consider for example a forest with 4 trees T_1, T_2, T_3, and T_4 in that order. Further, assume that T_1 and T_4 have depth 2 while T_2 and T_3 have depth 3. First, Treebeard reorders the trees to be in the order T_1, T_4, T_2, T_3. Then, the loop over the trees is fissed as shown in the following listing.

```
9   forest = ensemble(...)
10  for i = 0 to batchSize step 1 {
11    prediction = 0
12    for t = 0 to 2 step 1 {
13      tree = getTree(forest, t)
14      node = getRoot(tree)
15      node = traverseTreeTile(tree, node, rows[i])
16      treePrediction = getLeafValue(tree, node)
17      prediction = prediction + treePrediction
18    }
19    for t = 2 to 4 step 1 {
20      tree = getTree(forest, t)
21      node = getRoot(tree)
22      node = traverseTreeTile(tree, node, rows[i])
23      node = traverseTreeTile(tree, node, rows[i])
24      treePrediction = getLeafValue(tree, node)
25      prediction = prediction + treePrediction
26    }
```

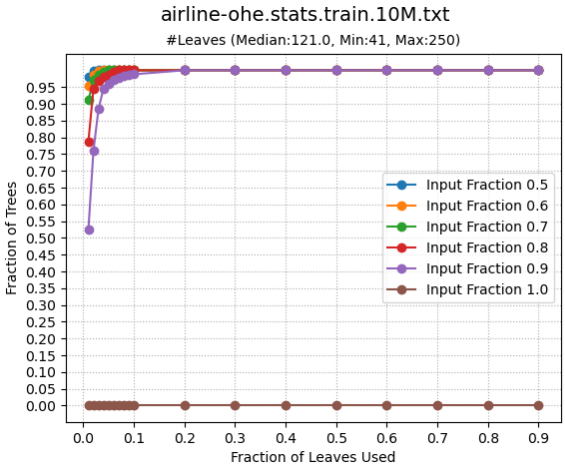

Figure 6: Statistical profile for airline-ohe

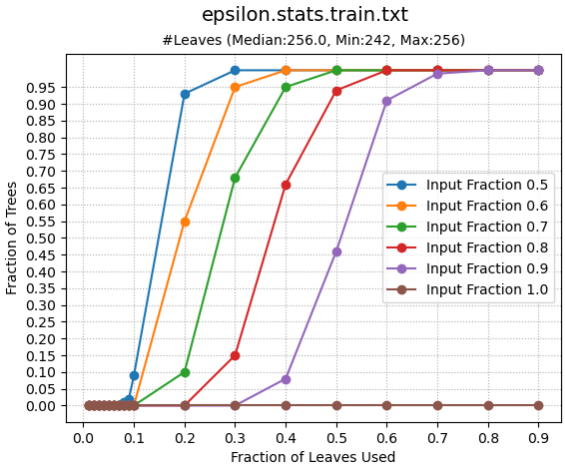

Figure 7: Statistical profile for epsilon

```
27    predictions[i] = prediction
28  }
```

TODO AP: This listing is unnecesarily long. Can we maybe leave out the loop bodies and say something like "depth 2 walk"?

3.10 Probability Based Tiling

3.10.1 Motivation

As discussed in section 3.9, uniform tiling optimizes tree walks assuming that all paths in the tree from root to leaf are equally likely. However, we find that this is not always the case. We performed inference on the complete training set of each of our benchmarks and measured how many times each leaf was reached. While for some benchmarks all leaves are roughly equally likely, for others, a small number of leaves are extremely likely while the rest aren't reached often. We call trees with a small number of extremely likely leaves ***skewed***. Figures 6 and 7 show this. Each line in these graphs corresponds to a fixed fraction of the input (say f). A point on a line at coordinate (x, y) means that y percent of trees in

the model could handle a fraction f of all training inputs with x percent of leaves. TODO Explain with a specific point in one of the graphs Figure 6 shows that very few leaves are reached for a very large fraction of inputs for the benchmark airline-ohe. This means that a small fraction of leaves are very likely. On the other hand, figure 7 shows that a large fraction of trees need a large fraction of their leaves to handle any fraction of the test input for the benchmark epsilon. One other observation we made is that most models have some skewed trees while the rest of the trees have equally likely leaves. TODO AP Define a term for trees with roughly equally likely leaves. We design the probability based tiling algorithm to take advantage of this property of decision tree ensembles. TODO This section needs to be rewritten. Define "leaves covering inputs" to clarify the presentation.

3.10.2 Notation

In order to formulate the probability based tiling algorithm as an optimization problem, we define the following.

1. For every leaf $l \in L$, we define p_l as the probability that the leaf l is reached.
2. For each node $n \in V$, we define the absolute probability p_v as
$$p_v = \begin{cases} p_l & \text{if } l \in L \\ p_{left(v)} + p_{right(v)} & \text{otherwise} \end{cases} \quad (2)$$
3. For any tree T, $\mathscr{C}(T)$ represents the set of all valid tilings of T.
4. For every $v \in V$, we define S_v as the subtree rooted at v.
5. For every $v \in V$, we define L_v as the set of leaves of S_v.
6. For a every tile T_i, we define $root(T_i)$ as the node $v \in T_i$ such that v has no incoming edges from any other node $u \in T_i$.
7. For a tile T_i, $out(T_i) \subseteq E$ is the set of edges (u,v) such that $u \in T_i$ and $v \notin T_i$.

3.10.3 The Optimization Problem

We observe that the latency of one tree walk is proportional to the number of tiles that need to be evaluated to reach the leaf. Therefore, the average inference latency can be modelled as the expected number of tiles that are evaluated to compute one tree prediction. More formally, the problem is to find a tiling $\mathscr{T}$ such that the following objective is minimized.

$$\min_{\mathscr{T} \in \mathscr{C}(T)} \sum_{l \in L} p_l . depth_{\mathscr{T}}(l)$$

where the minimization is over all valid tilings $\mathscr{T}$ of the tree T, $depth_{\mathscr{T}}(l)$ is the depth of the leaf l given tiling $\mathscr{T}$.

The constraints on the optimization are the constraints listed in Section 3.4. The above optimization problem can be solved optimally using dynamic programming. We leave this out in the interest of space. Instead, we use the simple greedy algorithm listed in algorithm 2 to construct a valid tiling given the node probabilities. The algorithm starts at the root and greedily keeps adding the most probable legal node to the current tile until the maximum tile size is reached. Subsequently, the tiling procedure is recursively performed on all nodes that are destinations for edges going out of the constructed tile.

Algorithm 2 Greedy Probability Based Tree Tiling

```
procedure TileTree(T = (V, E, r), n_t)
    if r ∈ L then
        return {r}
    end if
    Tile ← {r}
    while |Tile| < n_t do
        e = (u, v) ∈ Out(Tile) st p(v) is max and v ∉ L
        if e = ∅ then
            break
        end if
        Tile = Tile ∪ {v}
    end while
    Tiles = {Tile}
    for (u, v) ∈ Out(Tile) do
        Tiles ← Tiles ∪ TileTree(S_v, n_t)
    end for
    return Tiles
end procedure
```

When we tried to apply algorithm 2 on all trees in our benchmarks, we found that even minor variations in probability caused the tiling algorithm to generate a large number of tile shapes. This in turn caused a loss in performance because the large size of the lookup table needed (section 3.7) caused increased L1 cache misses. In order to alleviate this, we only perform probability based tiling on trees that are skewed. We say a tree is skewed if a small fraction of leaves, say α, cover a large fraction of training inputs, say β. We only perform probability based tiling on skewed trees and fall back to uniform tiling otherwise. In our experiments we use $\alpha = 0.05$ and $\beta = 0.9$.

3.10.4 Code Generation

As probability based tiling pulls the most probable leaves of a decision tree nearest the root, it poses some implementation challenges. By design, the tiling process makes the tree of tiles imbalanced. Therefore, the simple array based representation described in section 3.8.1 cannot be used because of the memory footprint blow up (a large part of the tree is empty, but would need to be allocated). On the other hand, the sparse representation in section 3.8.2 adds an extra hop for leaves that have non-leaf siblings. But this would mean that we add extra hops for the most probable leaves after probability based tiling. To address these problems, Treebeard peels the tree walk and specializes the leaf checks at higher levels to avoid the extra hop. Currently, we determine the maximum depth of leaves needed to cover 90 percent of the inputs and peel the tree walk by as many iterations. For example, consider the case where leaves until depth 2 are needed to cover 90 percent of the training input. Then, Treebeard generates the following IR.

```
29    // ...
```

```
    tree = getTree(forest, t)
    node = getRoot(tree)
    node = traverseTreeTile(tree, node, rows[i])
    if (isLeafTile(node)) {
        treePrediction = getLeafValue(tree, node)
    } else {
        node = traverseTreeTile(tree, node, rows[i])
        if (isLeafTile(node)) {
            treePrediction = getLeafValue(tree, node)
        } else {
            // Loop based traversal
        }
    }
    treePrediction = getLeafValue(tree, node)
    // ...
```

The if statements check whether a node is a leaf tile and hence avoid the extra hop. The memory requirement is also not increased because only a small fraction of leaves are represented as full tiles. While walk peeling is used to improve the performance of probability based tiling by specializing leaf tests, the transformation is by itself general and can be used in different contexts. For example, it could be used to elide leaf checks until a depth d is reached if we know all leaves are at a depth greater than d.

One other issue that the code generator needs to handle is that walks of different trees in the same ensemble may need to be peeled to different depths. In order to handle this, trees are reordered so that all trees with equal peeling depth are grouped together and the loops in the IR are fissed so that tree walks for these groups of trees can be specialized differently. This is very similar to the code generation strategy used for uniform tiling.

3.11 Unroll and Jam

While tiling and subsequent vectorization gives significant performance gains, profiling the generated code showed that true dependencies between instructions were still causing a significant number of stall cycles. In order to fill these stall cycles, Treebeard does an unroll and jam on the inner most loops of the loop nest. This transformation has the effect of walking multiple tree and input row pairs in an interleaved fashion. The dependency stalls can be hidden by scheduling instructions from independent tree walks in the stall cycles.

This optimization is performed across both the mid-level IR and low-level IR. Loops on which to perform unroll and jam are identified in the mid-level IR. This information is communicated to the lowering passes by replacing these tree walks with specialized tree walk operations. The lowering passes that transform the mid-level IR to low-level IR interleave operations across independent tree walks.

The following listing shows the mid-level IR when the inner loop over the input rows is unrolled by a factor of 2 and the two resulting tree walks are jammed together.

```
forest = ensemble(...)
for t = 0 to 4 step 1 {
  for i = 0 to 2 step 2 {
    tree = getTree(forest, t)
    prediction1, prediction2 = InterleavedWalk((tree,
   rows[i]), (tree, rows[i+1]))
  }
}
```

When lowered to low-level IR, the operations to traverse each of the tree, input row pairs (the arguments to the `InterleavedWalk`) are interleaved. One step of the interleaved walk is listed below.

```
// ...
n1 = n2 = getRoot(tree)
// ...
threshold1 = loadThresholds(n1)
threshold2 = loadThresholds(n2)
featureIndex1 = loadFeatureIndices(n1)
featureIndex2 = loadFeatureIndices(n2)
feature1 = rows[i][featureIndex1]
feature2 = rows[i][featureIndex2]
pred1 = feature1 < threshold1
pred2 = feature2 < threshold2
n1 = getChildNode(n1, pred1)
n2 = getChildNode(n2, pred2)
// ...
```

These transformations are fairly general and are not aware of the in memory representation of the model. Therefore, they are reusable across different in memory representations - the ones that are currently built into Treebeard or ones that are added in the future. TODO AP I feel the way this section is currently written makes the optimization seem extremely trivial. Is there a different way to present it?

3.12 Vectorization

Vectorization performed by Treebeard is enabled by the tiling transformations described in section 3.2. When the low level IR is translated to LLVM IR, Treebeard generates LLVM instructions that operate on the threholds and feature indices of nodes within a tile in a vector fashion. Therefore, thresholds and feature indices are loaded using vector loads and predicates are evaluated using vector comparisons. These vector LLVM IR instructions are then converted to vector instructions in the target ISA by the LLVM JIT.

3.13 Parallelization

Currently, Treebeard performs a naive parallelization of the inference computation. When parallelism is enabled, the loop over the input rows is parallelized using OpenMP. Treebeard rewrites the mid-level IR by tiling the loop over the input rows with a tile size equal to the number of cores. As a concrete example, consider the case where we intend to perform inference using a model with 4 trees on a batch of 64 rows. Further, assume that we wish to parallelize this computation across 8 cores. Treebeard then generates the following IR.

```
forest = ensemble(...)
parallel.for i0 = 0 to 64 step 8 {
  for i1 = 0 to 8 step 1 {
    i = i0 + i1
    prediction = 0
    for t = 0 to 4 step 1 {
      tree = getTree(forest, t)
      treePrediction = WalkTree(tree, rows[i])
      prediction = prediction + treePrediction
    }
    predictions[i] = prediction
  }
}
```

Currently, as our focus is on single core performance, we do not do any parallelism related optimizations (such as loop tiling). We leave a more thorough exploration of parallelization to future work.

REFERENCES

[1] C. Lattner and V. Adve, “Llvm: a compilation framework for lifelong program analysis amp; transformation,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.